Resilient, Unified, Shared Spaces In Ad hoc Networks

Daniel Cutting and Aaron Quigley
School of Information Technologies
University of Sydney
Sydney, NSW 2006, Australia
+61 2 9351 5711
{dcutting,aquigley}@it.usyd.edu.au

ABSTRACT
We introduce a light-weight middleware system based on tuple spaces and an accompanying distribution protocol (RUSSIAN) for building applications in wireless ad hoc networks. The system does not attempt to enforce strict semantics, instead offering "best effort" mechanisms for use by the applications. By tuning various protocol parameters, the system attempts to continually adapt these mechanisms as the contextual environment of the system changes.

1. INTRODUCTION

Wireless ad hoc networks are spontaneously formed by mobile devices with wireless communication capabilities. Unlike their fixed wireline system counterparts, they are particularly problematic for running distributed applications due to their unreliable and unpredictable behaviour. However their increasing importance as pervasive computing environments become more practical makes it necessary to consider how useful applications can be made to run reliably in such networks.

Although initial attempts to build applications adapted existing wireline techniques, such as RPC [7], these were found to be impractical in wireless ad hoc networks since they often relied on connection-oriented approaches that were unable to gracefully cope with frequent device disconnection and the severe resource limits imposed. Subsequent research has turned predominantly to middleware systems, and in particular, the Linda tuple space metaphor [4], which has been used as the basis for several middleware systems aiming to support distributed applications in wireless ad hoc networks [9, 11].

The use of contextual information available in the system has been used to try to develop more adaptive middleware systems [3, 8]. It is proposed that by making middleware systems cater specifically to the current situation of a network, they can more efficiently and reliably service the applications they support.

2. AIM

Instead of trying to deal with all possible configurations of pervasive computing environments that mix both mobile devices and fixed infrastructure, we concentrate specifically on small, localised ad hoc networks comprising only resource-constrained mobile devices such as PDAs and smart phones.

Even with this limitation, it is possible to think of many distributed applications that may be desirable for these environments including file sharing tools, chat rooms, multi-player games and business utilities. It is worth noting that these applications may not generally be too demanding in terms of strict coordination. For example, it is not absolutely essential that every message entered into a chat room be delivered to all participants immediately after it is entered (and in some cases, it is even acceptable for messages to be lost, etc). Similarly, in non-action multi-player games, it is generally acceptable for player's views to be slightly inconsistent, and if it takes five seconds instead of one second to discover that a new file is available in a file sharing application, this does not generally mean the application is useless.

Our overall goal is to support the sharing of high-level data for such applications in a manner that requires no configuration with an extremely light-weight middleware system with minimal resource requirements. Unlike some other systems that attempt to strictly enforce application coordination at the middleware layer, we feel this should instead be moved to the applications and supported by "best effort" mechanisms from the middleware.

Also, unlike many other mobile middleware systems, we feel it is important for the middleware to be truly spatially and temporally decoupled. To this end, data inserted into the middleware should be made available as long as possible, even when the producer of that data is permanently removed from the system. It is expected that such data resilience will allow better handling of unexpected disconnection and failure of devices.

3. APPROACH

3.1 Model

As with other middleware systems designed for such environments [3, 9, 11], we use the Linda tuple space metaphor as the means of enabling communication between application components on different devices.

Linda [4] is a coordination language for easily distributing applications and has been implemented both as an extension to many languages and as distinct systems. In essence it uses a global, shared tuple space paradigm and defines three operators for accessing the space. A tuple is a simple, ordered, record-like data structure of the form $< a, b, ... >$, and a tuple space can be thought of as a shared bag (or multiset) of such tuples. Tuples can be constructed from both *actual* and *formal* elements, where actuals are literal values, and formals are types. An *anti-tuple* or *template* is a tuple that can be used to match against another tuple. Matches are made

when the number of elements of two tuples are equal, all actual elements are equal and all formal elements match the type of corresponding actuals. The basic operators in Linda are:

- OUT(t) places the tuple t into the tuple space.
- IN(t) withdraws a tuple matching the anti-tuple t from the tuple space. This operator blocks until a match is found. The returned tuple is guaranteed to be immediately removed from the tuple space and no copies will be given to any other process.
- READ(t) copies (without withdrawing) a tuple matching the anti-tuple t from the tuple space. This operator blocks until a match is found.

In addition to these, EVAL(t) spawns a new process to evaluate the elements of t before it is placed into the tuple space.

Since all communication between processes is achieved via these operators, it is not necessary for any process to know about the operation, name, or even existence of any other. This makes the processes anonymous and spatially decoupled. Furthermore, since tuples can permanently reside in tuple space until they are withdrawn with an IN operation, regardless of whether the process that created them is still running, the processes are temporally decoupled.

The spatial and temporal decoupling of components and simple operators afforded by tuple spaces are desirable properties in wireless ad hoc networks since such networks are formed from devices that may never have interacted before, and the membership of the network changes frequently.

However, it is difficult both to ensure correct application execution and to maintain the original Linda semantics of a unified tuple space in situations where devices comprising the group regularly fail or become disconnected. The failure or disappearance of a device constituting part of a tuple space has repercussions at both the application and system level. For example, if an application has been designed to rely on a shared "token" tuple that is held by a device at the time of its failure, there is little the system can do to help the application recover beyond providing as much context and information to the application as possible - it is up to the application to resolve the issue and move forward (see section 4 for some alternatives). At a lower level however, the system must be able to cope with device failures in a graceful way that does not unnecessarily affect the applications. For example, the disappearance of a device that is not actively participating in the application should not cause the application to fail. Similarly, the failure of a device that has inserted some useful data into the tuple space should not make that data unavailable in general.

Instead of trying to maintain the original Linda semantics in such circumstances, it may be more appropriate to relax the model. In order to address the application level issue, we relax the original Linda operators by adding timeouts to the blocking operators IN and READ. This prevents any component from deadlocking in the event of a device failing and losing an important tuple. In order to aid the application in determining why the timeout has occurred, all contextual information maintained by the middleware system is also available to the application.

To reduce system level problems, data resilience is improved by caching tuples around the network and allowing these to be returned in cases where the original owner is no longer part of the system (the precise realisation of this is described in the following sections). However, by caching tuples, we introduce the possibility of cache inconsistency, and thus tuple space inconsistency, which is reflected in the relaxed IN operator described below and discussed further in the following sections. The operators in the relaxed model are thus:

- OUT(t) places the tuple t into the tuple space. It is not guaranteed that the tuple will be visible by all participants in the space immediately, though its availability should improve over time as it becomes cached around the network.
- IN(t) retrieves a tuple matching the anti-tuple t from the tuple space. This operator blocks until a match is found or a timeout occurs. Unlike Linda, the tuple will not necessarily be removed from the tuple space immediately and it may be possible for other devices to retrieve it simultaneously, though the system will make a "best effort" to reduce these risks.
- READ(t) copies a tuple matching the anti-tuple t from the tuple space. This operator blocks until a match is found or a timeout occurs. It may also return a stale tuple (one that has recently been withdrawn from the tuple space but for which replicas remain). This is acceptable because any data returned to the application from a READ is out of date the moment it is received anyway, as the tuple space could have changed before the application acts on the information.

Although these modifications may seem to negate the usefulness of tuple spaces for coordinating applications, they still provide simple mechanisms for sharing data between properly designed application components.

With the tuple space operators thus defined, it is important to understand how interaction between devices will actually take place. Although we have so far concentrated on a single tuple space, the use of multiple tuple spaces is not a new idea [5]. It is useful for partitioning unrelated sets of tuples and can improve performance when it is known how the tuple space will be used. Often, new operators are introduced to create and destroy tuple spaces but for our model, we simply assume that tuple spaces are eternal and need not be explicitly reified. To access a tuple space, a device simply invokes its name (a unique identifier of some sort). If multiple devices agree on the same name (perhaps discovered through some sort of service discovery protocols), they will automatically access the same tuple space.

As a device leaves the proximity of other devices using the same tuple space, it will take with it a (probably incomplete) set of cached tuples. From its point of view, this set will then constitute the whole tuple space. If it should later encounter other devices using the same space, the tuples will be made available to those devices (and vice versa). It is thus pos-

sible for a device to automatically act as a physical means of migrating tuples between different sets of devices. There is of course the problem of the device modifying the tuple space and reencountering the initial set of devices, which have also modified their version of the tuple space. In situations like this, where there will be cache inconsistency problems, the system employs mechanisms for converging the disparate views over time, described more fully in the following sections.

3.2 Realisation

The problem of physically distributing tuples in the tuple space over the devices that comprise the network can be complex.

In wireline situations, it is relatively straightforward to distribute a tuple space over many machines since several assumptions can be made. In particular, it can usually be assumed that devices will not fail unexpectedly and it can often be assumed that communication between hosts is quite fast. There are two immediately apparent, straightforward distribution algorithms when making such assumptions. The first broadcasts each OUT tuple to all hosts in the system where it is cached. READ and IN operations are quite cheap since a complete copy of the tuple space exists on each host. (Of course, an IN operation also requires a protocol for removing the copy of the matched tuple from all other hosts.) The second algorithm is almost the opposite. An OUT operation stores the tuple locally, but an IN or READ operation triggers the searching of all hosts for matches. There are more sophisticated approaches that fall somewhere in between these extremes. Instead of simply broadcasting OUT tuples, [1] uses a hash function to determine upon which hosts to store them. The same hash can be used to return a list of servers where the tuple could exist when a (partial) template is given. This dramatically improves performance when the system comprises many hosts.

However, wireless ad hoc networks have properties quite unlike their stable, wireline counterparts. Specifically, it is very common for new devices to join and leave the network unannounced, and the wireless nature of communications means it is comparatively easy for messages between devices to go unheard. These drawbacks mean that tuples exclusively stored on single devices could be lost or made unavailable for long periods. Although these problems could potentially be remedied by using the broadcast approach mentioned above, this would unnecessarily use a large amount of power in a system where power is at a premium.

Our approach to the problem of distributing tuples is based on several assumptions:

- In a small, localised wireless ad hoc network, multicasting (or broadcasting) messages is as cheap as unicasting since all devices are within one hop.
- Since devices can physically come and go at any time, and hence become frequently disconnected from the network, it is pointless to use "reliable" communication protocols (i.e. protocols that try to guarantee delivery of messages via acknowledgments, etc.); devices will need to be able to cope with unreliable peers anyway.
- The applications to be deployed in wireless ad hoc networks will not typically require strict semantics and guarantees (since there can be no guarantee that necessary data or devices will be part of the network anyway).

Linda tuple spaces are inherently spatially and temporally decoupled. This means that an inserted tuple should no longer be tied in any way to its producer. Many distributed tuple spaces have modified this feature, by requiring a tuple be "owned" by one participant in the system [3, 9, 11]. Since there is no server in a fully distributed system, this concept of ownership allows a single device to decide which requests for a particular tuple should succeed and which should fail. A drawback of the approach is that the tuple is no longer truly spatially or temporally decoupled from the system; if the device that owns the tuple fails or becomes disconnected, the tuple is no longer available to other devices.

The realisation of our middleware system relies on two main concepts. Firstly, whenever devices need to communicate, the messages should be broadcast to all devices so that they can have a mostly complete impression of the system state. This is an extremely important point, as it means that all devices are free to respond to any message at any time, and tuples overheard in transit can be cached to satisfy future requests. Secondly, at any one time a single device should act as an *arbitrator* and make rulings in situations as required by tuple space semantics. The arbitrator is our approach to handling the problems arising from having no concept of ownership in the model. The arbitrator can be thought of as temporarily owning tuples when their ownership is in dispute.

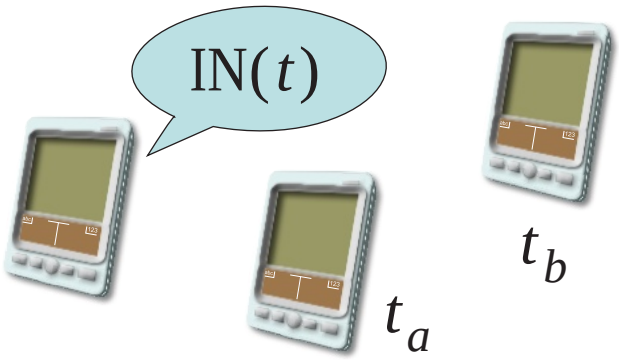

Figure 1: The multiple match problem. Which tuple (t*a* or t*b*) should be returned and removed from the tuple space?

The purpose of an arbitrator can be illustrated with some sample scenarios, shown in figures 1, 2 and 3. Figure 1 shows a device performing an IN operation. The two nearby devices each hold different tuples that could satisfy the request. The problem is agreeing on which tuple is returned and removed from the tuple space. The problem can be solved by both devices broadcasting their tuple, having the requester choose one arbitrarily, and broadcast its choice. Both devices would thus know which tuple had been selected and whether their tuple was still available to satisfy other requests. In this case, the arbitrary choice is made by the requester itself, but since all messages are heard throughout the network, any device could have made this decision and

acted as the arbitrator.

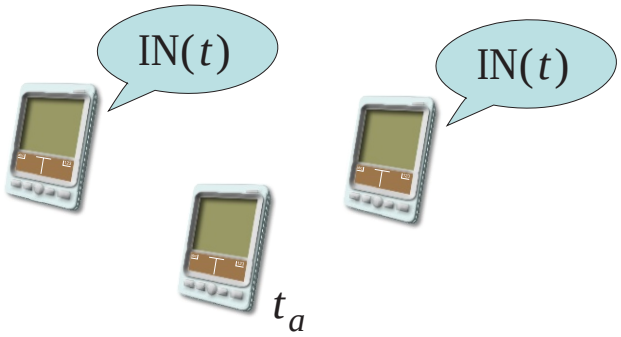

Figure 2: The “at most one” problem. Which device should succeed and get the tuple t*a*?

Figure 2 shows a simpler situation. If a device hears two virtually simultaneous requests that can both be satisfied by its only tuple, it needs to select a victor. This could be solved by it arbitrarily choosing a victor and broadcasting the tuple and a message to that effect. However, since this is a special case, where no other devices happen to have tuples matching the request, it can be subsumed into the first scenario, outlined above. Since both requesters would hear the broadcast, and since other replies may have been broadcast by other devices, it can be the responsibility of the arbitrator to make the final decision.

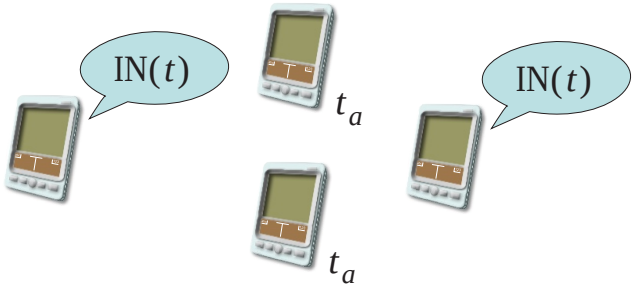

Figure 3: The cache problem. When cached copies of the same tuple exist and simultaneous requests are made that match that tuple, it’s possible for each to be satisfied by different cached copies.

Figure 3 shows a more difficult problem. Since the model replicates tuples on different devices as much as possible to improve data resilience, it is possible for different copies of the same tuple to satisfy separate destructive requests. In this case, it is not sufficient for the holder of a copy and a requester to agree (unlike the scenarios above where no cached copies exist), since this could happen simultaneously throughout the network, against the general semantics of the IN operator. In this case, an arbitrator (which should have heard both requests) can act as the centralised point that makes the decision of which request actually succeeds. Upon hearing the arbitration, each requester knows whether they succeeded or not, and the copy holders can both remove the tuple from their caches.

The concept of an arbitrator allows the system to epidemically cache tuples on as many devices as desired, since the arbitrator can act as the centralised point for enforcing semantics. This is, of course, a bottleneck on the system. With a sufficiently intelligent algorithm however, it should be possible to have different arbitrators for different sets of tuples, decided on the basis of a tuple hash, for example. Another problem arises when the arbitrator fails or loses connectivity. The model is able to recover from this situation by having another device take over the role after a period when the arbitrator does not seem to have replied. Since all devices should hear all (or most) messages, this is relatively straightforward.

3.3 Protocol

The “Resilient, Unified Shared Spaces In Ad hoc Networks (RUSSIAN)” protocol is an unreliable, multicast datagram protocol for distributing tuples over multiple devices. We provide here a general overview of the protocol and an informal description of its operation. As outlined above, it is assumed that all devices participating are generally within broadcast range of one another and can thus potentially overhear all messages.

3.3.1 Initiating an operation

An OUT operation can either result in no network access or the broadcast of the entire tuple. It is not important to the algorithm which choice is made by a particular device, since IN and READ requests are usually broadcast anyway (see below). An advantage of broadcasting the tuple is that all devices that hear the broadcast can cache it to satisfy future requests. In many cases however, this would waste resources. The decision to broadcast or not is influenced by the context engine (see section 3.4).

When a device wants to perform an IN operation, it should broadcast a request and wait for a timeout, or a reply followed by an arbitration message. Any device holding a matching tuple can reply, though if no replies are heard and/or no arbitration message is heard, the operation fails after a timeout. If the arbitration permits usage of the tuple, it is passed back to the application, otherwise the IN operation waits for other possibilities or times out.

When a device wants to READ a tuple, it should first check its own caches. If no match is found, a request should be broadcast and the device should wait for a timeout or a reply. Any device that has a matching tuple can reply. No arbitration is generally required for a READ, as there are no contention issues over who should receive the tuple as with IN.

3.3.2 Listening to broadcasts

When a device hears a full tuple being broadcast as an OUT operation or a reply to a READ or IN operation, it can choose to cache it or not according to the policy on the device. If the tuple has been broadcast in response to an IN, it should only be cached if the IN is known to subsequently fail (according to an arbitration message). If a device caches a tuple, it should verify if any pending READs or INs it has overheard can be satisfied.

When a device hears a READ or an IN operation, it should

check its caches for possible matches. If one is found, it should wait a small random time before broadcasting it. If while waiting it hears another device replying, it should cancel its reply and ignore the request. The amount of time before replying can be tuned according to the context engine (see section 3.4). Devices should probabilistically reply more quickly when they themselves have OUTed the matching tuple and reply more slowly when replying with a cached tuple. In this way, fewer messages are likely to be sent overall, since most requests will be satisfied by just a single device. If the producer of the original tuple is no longer available however, the request can still be satisfied by a cached copy.

When a device overhears an arbitration message intended for another device, it should remove any cached copies of the now consumed tuple so they are not reused in future operations.

3.3.3 Handling cache inconsistencies

There will be times when a device loses contact and does not hear that a cached tuple has been withdrawn. When it reconnects it will respond to requests with that invalid tuple. This can be mitigated with a "withdrawn list" maintained by all devices. If a device offers up a cached copy that is known by other devices to have been withdrawn, they can broadcast a veto. The arbitrator can use this veto to deny the request, and all devices that hear the veto can update their caches. However, this would imply the arbitrator would always need to wait for vetoes when handling an arbitration, which is highly inefficient. To prevent this, replies to IN operations could be marked as "certain" if they were created by the device that is replying as opposed to being a cached copy. The arbitrator can be certain in these cases that the tuple is current and immediately broadcast the arbitration. It is only when no certain replies have been offered that the arbitrator must consider waiting for vetoes. Lists of recently withdrawn tuples could be piggybacked onto other broadcast messages to improve cache consistency without additional messages.

3.4 Dynamic adaptation

The RUSSIAN protocol outlined above has several tunable parameters that can greatly affect the efficiency of the system as a whole and the resilience of the data stored in the tuple spaces:

- Whether an OUT operation should do nothing or broadcast the entire tuple. Obviously broadcasting a tuple is prohibitively expensive if no other devices are likely to be interested, but it also means tuples will probably survive the failure of the device.
- How long a device should wait before broadcasting replies to IN and READ operations. Although a device should generally prefer broadcasting tuples it created to those it has cached from other devices, it may become clear over time that it is holding the only copies of tuples from a failed device and should broadcast them as if it had created them, for example.
- Whether (and which) overheard tuples should be cached. OUT operations and replies to IN and READ operations will include full copies of tuples that can be cached to satisfy future requests but whether or not these should be cached can be determined by such variables as remaining storage space and metadata attached to the tuple which may indicate how long the tuple is expected to be needed or how important it is.

These parameters may be dynamically tuned by a context engine that aggregates several sources of contextual information including device characteristics, application usage patterns, apparent device appearance and disappearance (determined by monitoring messages from devices), and perceived stability within the network. Internal modeling of the overall system state would be a useful tool for determining how the tuning should proceed, and since all messages in the system are available to each device, quite sophisticated models should be possible. Clearly, smaller devices without storage capabilities or slow processors will be less able to perform this modeling and appropriate defaults can be preset.

Furthermore, it is not necessary for each device to base its model purely on the messages overheard in the system; additional contextual information generated by devices could be supplied to others in a sub-protocol piggybacked onto the RUSSIAN protocol, which they could use to update their own model. The system model would also help a device determine whether it is an appropriate choice for arbitrator should an existing arbitrator fail.

4. RELATED WORK

Limbo [11] is an initial attempt to use the tuple space metaphor in a mobile computing environment. It is argued that the spatial and temporal decoupling of processes in Linda maps well to the nature of mobile networks but concludes that tuple spaces generally do not lend themselves well to measuring network quality of service (precisely because they are connectionless). Limbo uses a multicast protocol to distribute tuples over multiple devices, but uses the concept of ownership of tuples (and transferral of that ownership) to enforce the semantics of the Linda IN operator. This approach means that the owner of a tuple must be connected to the network whereas in the RUSSIAN protocol, which uses tuple replication and has no concept of ownership, any device that has a cached copy is free to return the tuple.

LIME [9] is another implementation of Linda in a mobile environment that modifies the notion of a tuple space considerably and supports both physical (through the environment) and logical mobility of agents (from host to host). Instead of a single, unified tuple space, spaces are partitioned and merged. Each agent has its own tuple space which is merged with those of other agents residing on the same host. When hosts come within range of one another, their tuple spaces are automatically merged and separated when they move out of range. A useful mechanism is also provided for triggering reactions when a tuple matching a subscription template is inserted into the tuple space. As with Limbo, LIME uses a concept of ownership of tuples to maintain Linda semantics.

Limone's [3] approach is to have each device maintain an acquaintance list which enumerates nearby available peers with which it can communicate. This list, which is man-

ageable from the application level, is continually updated by each device transmitting beacon signals. Instead of a unified, shared space, each agent maintains and uses its own space. When it wishes to receive data from a peer, it requests it to perform a tuple space operation on its own space and return the results. This approach is quite different from the original Linda tuple space metaphor, removing some of the anonymity and temporal decoupling in that design. However, it makes possible more predictable and semantically rigid communication between application components. The RUSSIAN protocol has deliberately moved in the opposite direction, opting instead to try and maintain the unification of the tuple space above the enforcement of strict Linda semantics. Our system also foregoes expensive beaconing. Since each request is potentially satisfiable by any host there is no particular advantage in knowing which hosts are currently nearby.

There have also been fault-tolerant techniques to handle host failure in more conventional distributed Linda systems. Some systems have used the notion of transactions to ensure that when failures occur the applications are able to rollback actions and continue. Though somewhat heavyweight, this approach is quite popular in some tuple space systems such as JavaSpaces [6]. Transactions are seen to be too expensive for our system as our goal is to make it extremely light-weight for resource-constrained devices. An alternative approach to transactions is the idea of *agent wills* [10]. It is argued that transactions are too expensive in many situations and that a mechanism allowing a host to clean up an operation if it fails during its execution is sufficient. This concept however, relies on the ability to migrate an atomic unit of code for execution on a remote host and is inappropriate in heterogeneous systems.

5. FUTURE WORK AND CONCLUSION

Our research aims to provide a very light-weight middleware system for distributing applications in wireless ad hoc networks. Recognising that the style of application we wish to deploy does not usually require strict operational guarantees, we do not attempt to enforce strict semantics, instead offering "best effort" mechanisms for use by the applications. By tuning various protocol parameters, the system attempts to continually adapt these mechanisms as the contextual environment of the system changes.

It remains to be seen whether the RUSSIAN protocol is resilient enough to provide a useful service to distributed applications in wireless ad hoc networks. As such, we intend to implement the protocol and test it in a wide variety of simulated networks, especially in cases where network quality is very poor and devices move in and out of range at a rapid pace. In addition to low-level simulations, it would be illuminating to implement real applications for qualitative testing.

The model so far provides only the most fundamental operators necessary for a tuple space system, namely OUT, IN and READ. It has been argued that mobile computing platforms would benefit from the notion of events triggered by the insertion of data [9, 2], much like typical publish-subscribe middleware systems. Incorporating events into the RUSSIAN protocol may not be especially difficult, since it can be seen as a special case of a READ operation, though it would certainly require extra effort to maintain any useful semantics for such a feature.

ACKNOWLEDGMENTS

The authors would like to acknowledge the ongoing support of the Smart Internet Technology CRC and NICTA, and Darrall Cutting for his proofreading advice.

REFERENCES

[1] R. Alvez and S. Yovine. Distributed implementation of a linda kernel. In *Proceedings of XVII Conf. Latinoamericana de Informatica*, 1991.

[2] G. Cugola and E. D. Nitto. Using a publish/subscribe middleware to support mobile computing. In *Proceedings of the Workshop on Middleware for Mobile Computing, in association with IFIP/ACM Middleware 2001 Conference*, Heidelberg, Germany, November 2001.

[3] C.-L. Fok and G.-C. Roman. A lightweight coordination model and middleware for mobile computing.

[4] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[5] D. Gelernter. Multiple tuple spaces in Linda. In E. Odijk, M. Rem, and J.-C. Syre, editors, *Parallel Architectures and Languages Europe (PARLE '89)*, pages 20–27, Eindhoven, The Netherlands, 1989. Springer-Verlag.

[6] JavaSpaces(TM) service specification. Online. http://java.sun.com/products/jini/2.0/doc/specs/html/jsTOC.html.

[7] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek. Mobile computing with the Rover Toolkit. *IEEE Transactions on Computers*, 46(3):337–352, 1997.

[8] C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich. XMIDDLE: A data-sharing middleware for mobile computing. *Wirel. Pers. Commun.*, 21(1):77–103, 2002.

[9] A. L. Murphy, G. P. Picco, and G.-C. Roman. LIME: A coordination middleware supporting mobility of hosts and agents. Technical Report WUCSE-03-21, Washington University, April 2003.

[10] A. Rowstron. Mobile co-ordination: Providing fault tolerance in tuple space based co-ordination languages.

[11] S. P. Wade. *An Investigation into the use of the Tuple Space Paradigm in Mobile Computing Environments*. PhD thesis, Lancaster University, 1999.